Uso del simulador SimpleReuse.

C. Álvarez M. Valero

November 11, 2005

Abstract

Este report explica el funcionamiento del simulador SimpleReuse (una evolución de SimpleScalar y Wattch) para su uso en simulación de reuso de instrucciones o regiones. Este simulador realiza medidas de tiempo, instrucciones y consumo de potencia de códigos Alpha ejecutados en máquinas de muy distinta complejidad. El simulador no se encuentra disponible en la Web en este momento debido a que está en constante mejora, pero si hay alguien interesado en su uso puede pedirlo directamente a través de la dirección: carlos.alvarez@upc.edu.

1 Introducción

En los últimos años la proliferación de los sistemas de baja potencia (móviles, PDAs, etc.) ha generado una enorme demanda de capacidad de cálculo hacia estos. Los sistemas móviles de tercera generación se espera que sean capaces de ejecutar aplicaciones multimedia que hasta ahora estaban restringidas a los sistemas de propósito general de sobremesa. Dichas aplicaciones (codificadores y decodificadores MP3, reproducción de vídeo, reconocimiento de voz, etc.) poseen unos requerimientos de capacidad de cálculo muy superiores a los que actualmente es capaz de proporcionar un procesador de baja potencia. Por ello los sistemas se baja potencia han de enfrentarse al difícil reto de mejorar sus prestaciones de cálculo (incluyendo muchas veces el cálculo en punto flotante) manteniendo limitado su consumo.

En este ámbito, el cálculo difuso se propone como un sistema que permite obtener mayores velocidades de cálculo con bajo consumo a cambio de pequeñas perdidas de calidad en las señales finales generadas. Esta aproximación cobra sentido en el ámbito de las aplicaciones multimedia. Dichas aplicaciones se caracterizan por el hecho de que su destinatario final es una persona y por lo tanto la calidad de su salida depende de un criterio subjetivo. En este entorno pequeñas variaciones de los valores de salida no son percibidas y así pues podemos aprovechar esta circunstancia para conseguir beneficios adicionales. Un ejemplo claro de esto lo encontramos en los sistemas de compresión con perdidas como JPEG o MP3. En dichos sistemas, pequeñas perdidas de calidad en las señales se usan para conseguir mayores ratios de compresión. Las perdidas

no sólo son aceptables sino muchas veces imperceptibles para el usuario mientras que las ganancias en cantidad de información almacenada son significativas.

Para poder medir los efectos de los sistemas de cálculo difuso en programas reales ha sido necesario "ejecutar" dichos programas en un sistema que implementara el cálculo difuso, para ello, se ha debido construir un siulador que, basado en una arquitectura real, añadiese a esta las características propias del sistema a estudiar. En este report se explica el funcionamiento de este simulador que se ha denominado "SimpleReuse".

2 El simulador SimpleReuse.

El simulador SimpleReuse se ha elaborado a partir de los simuladores SimpleScalar[1] y Wattch[2] como el simulador de trabajo de la Tésis "Fuzzy Computation on Multimedia". Este simulador ha partido de la necesidad de elaborar estudios de la viabilidad de realizar computación difusa en las aplicaciones multimedia. Para ello, los simuladores de origen, tenían las siguientes limitaciones:

- No realizaban detalladamente los cálculos del consumo de las unidades aritméticas.
- No permitían realizar cálculos a medida, es decir, todas las operaciones aritméticas se realizaban usando el procesador de la máquina que ejecutaba el simulador y, por tanto, no era posible estudiar sistemas alternativos de realizar las operaciones y evaluar sus resultados.
- No tenían integrado un sistema de configuración detallado. Es decir, su integración permitía estudiar la potencia consumida por una máquina tipo Alpha (la máquina principal que simula SimpleScalar) pero, así como SimpleScalar admite otras configuraciones, Wattch no admitía dichas configuraciones en su estructura y por lo tanto no permitía cálculos de consumo de energía basados en diferentes ficheros de configuración.
- No disponían de un sistema que permitiese realizar las operaciones aritméticas en diferentes unidades funcionales y medir sus efectos.
- No entraban en el detalle del consumo de las operaciones aritméticas de punto flotante.

Dadas todas estas limitaciones enumeradas, se decidió elaborar un nuevo simulador integrado que, partiendo del sistema integrado de SimpleScalar y Wattch, fuese operativo a todos estos niveles. El simulador resultante de bautizó como SimpleReuse.

3 Uso del simulador.

El simulador resultante se ejecuta a traves del compando `sim-reuse`, que es una versión incrementada del programa `sim-outorder` de SimpleScalar, con

Wattch integrado y todas las modificaciones realizadas. Como ya se ha comentado, Wattch ha sido totalmente integrado, de forma que no es necesario editar el simulador Wattch y recompilarlo para aceptar nuevas configuraciones, sino que directamente este simulador activa y desactiva los componentes de Wattch necesarios (por ejemplo, si no disponemos de memoria caché, el componente de consumo de `sim-reuse` no computa gasto de energia por la caché).

Así pues, para poder usar el simulador simplemente hemos de tener los ficheros de configuración adecuados para ello. Dichos ficheros son:

- El fichero de configuración de SimpleScalar. Dicho fichero se indica en línea de comandos mediante la opción `-config`. Sus opciones son las mismas que se indican en la versión original del programa, pero han sido integradas de forma que el consumo de potencia se adapta REALMENTE a lo que indica el fichero. En el apéndice B se muestran varios ejemplos de estos ficheros. Todas las opciones de este fichero se podrían especificar también en línea de comandos, pero como son muchas esta posibilidad resulta incómoda.

- El resto de ficheros de configuración son específicos del sistema de reuso de instrucciones. El fichero principal (que tiene nombre único, no configurable) es el fichero `ISA_Gral.conf`. Si dicho fichero no existe el simulador genera un mensaje de error y acaba su ejecución. Este fichero contiene dos valores numéricos (habitualmente 0 o 1). El primero indica si está activado el reuso de regiones y el segundo si está activado el reuso de instrucciones. Si ambas opciones están desactivadas el simulador, simplemente, simulará un sistema sin reuso (pero seguirá contando bien el consumo de los elementos activados o desactivados mediante el fichero indicado en -config.

3.1 Reuso de regiones.

En la opción de reuso de regiones, los ficheros utilizados por el simulador son:

- En el reuso de regiones, el primer fichero de configuración, cuyo nombre por defecto es `ISA_bitr.conf` contiene el número de bits que se perderán en cada región reusada. Consta de una sola línea de texto que contiene un número entero para cada región distinta a reusar (conviene recordar que una región distinta implica una secuencia de operaciones distinta y por lo tanto diferentes regiones no pueden compartir resultados ni han de compartir necesariamente parámetros).

- El segundo es el fichero `ISA_tabr.conf`. Este fichero contiene 3 parámetros que indican, respectivamente, la cantidad de regiones distintas, el logaritmo en base 2 de la longitud de la tabla de regiones y la asociatividad de dicha tabla.

Las regiones a simular en este entorno se definen manualmente sobre el programa simulado. Para ello se han reinterpretado instrucciones del procesador

de Alpha que se encontraban hasta ahora sin uso y que el compilador no utiliza. El codigo de los programas a estudiar debe ser modificado y recompilado usando estas instrucciones (que se pueden insertar en el código C mediante la directiva `asm` si se incluye la librería `c_asm.h`):

- `excb`. Esta instrucción activa y desactiva el simulador de reuso de regiones. Dentro de este modo el resto de instrucciones se comportan como se describe, fuera de él se comporta como lo harían normalmente (lo que en la mayoria de casos significa que no hacen nada ya que no se encontraban implementadas en el simulador).
- `wmb`. Esta instrucción genera un acceso a la tabla de reuso.
- `mb`. Esta instrucción inicializa las tablas de reuso de regiones.
- `bne`. Esta instrucción, dentro de la simulación del reuso de regiones, no salta cuando ha habido un acierto en la tabla. Sirve para implementar los dos caminos posibles que ocurren después de acceder a la tabla.
- `bisi`. Esta instrucción, dentro de la simulación del reuso de regiones, sirve para pasar los parámetros hacia la tabla de reuso de regiones. El primer valor es el parámetro de entrada y el inmediato es el número de bits a descartar de dicho parámetro. Si la tabla ya ha sido accedida, en cambio, lo que hace es asignar los resultados.
- `trapb`. Esta instrucción actualiza la tabla de regiones despues de un fallo.

Mediante estas instrucciones ensamblador es posible modificar cualquier código para crear una región reusable. El código se modifica de la siguiente manera. Imaginemos que las instrucciones que queremos sustituir son las siguientes:

```
if ((v = t0  - t1)<0)  v = -v; res = v;
if ((v = t2  - t3)<0)  v = -v; res += v;
```

El código que deberemos insertar es:

```
   asm("excb ");
      res=t0|1; res=t1|1;
      res=t2|1; res=t3|1;
      asm("wmb");
      if (res) goto seguirR1;
      res=1;
   asm("excb ");
   goto pruebaR1;
seguirR1:

if ((v = t0  - t1)<0)  v = -v; res = v;
if ((v = t2  - t3)<0)  v = -v; res += v;
```

```
   asm("excb ");
     res=res|1;
     asm("trapb");
   asm("excb ");
pruebaR1:
```

Como se puede ver el código insertado tiene dos partes, una previa al código a reusar y otra posterior. La parte previa inicializa la tabla, y realiza el acceso. La parte posterior actualiza la tabla en caso de que hubiera un fallo. Vamos a comentarlo:

La primera instruccion (`excb`) activa el modo de reuso de regiones del simulador. A continuación tantos OR (que se traducen como la orden `bisi`) como sean necesarios cargan en la tabla los parámetros de entrada. Fijémonos que en nuestro código, todo depende de los valores `t0` a `t3`, así que cada instrucción OR carga uno de estos parámetros de entrada en la tabla. El 1 (parámetro inmediato de la instrucción OR) dice que toleraremos 1 solo bit de cada parámetro, mientras que el destino de la operación no se usa para nada.

Una vez cargados los parámetros de acceso, la instrucción `wmb` realiza el acceso a la tabla y determina si ha habido un fallo o un acierto. La siguiente orden (el `if`), contra lo que pueda parecer, solo saltará si ha habido un fallo en el acceso (recordemos que la instrucción `bne` ha sido puenteada), así que si ha habido fallo se continuará a partir de la etiqueta `seguirR1`. Si ha habido un acceso a la tabla, las instrucciones siguientes al if asignan a las variables resultantes (en el ejemplo solo la variable `res`) el valor devuelto por la tabla (la isntruccion `res=1` también queda codificada mediante la instrucción ensamblador `bisi`). En ese caso, después de realizada la asignación se desactiva el modo de simulación de regiones (siguiente instrucción `excb`) y se continúa la ejecución a partir de la etiqueta `pruebaR1`, de forma que se saltan las instrucciones de la región reusada.

Si no ha habido acierto en la tabla, la región a reusar se ejecuta normalmente y a continuación se vuelve a activar el modo de reuso de regiones para actualizar la tabla de reuso. Mediante la primera instrucción se carga el resultado (o resultados) en la tabla y mediante la segunda (`trapb`) se actualiza la última entrada accedida de la tabla. El último `excb` desactiva de nuevo el modo de reuso de regiones.

3.2 Reuso de instrucciones.

La simulación de reuso de instrucciones, al contrario que la de regiones, no precisa ninguna modificación de los programas originales. Estos simplemente, una vez compilados, se ejecutan bajo el simulador y este genera los distintos resultados según su configuración. Los resultados de prueba pueden obtenerse, o bien mediante la ejecución de los programas bajo la máquina original, bien bajo el simulador desactivando el sistema de reuso.

Los ficheros de configuración para el reuso de instrucciones son:

- `ISA_act.conf`. Este fichero (cuyo nombre se puede modificar mediante la opción `-f:act`) contiene una lista con un valor para cada posible instruccion en la que se puede activar el reuso. Dicho valor es verdadero o falso según si se reusa dicha instruccion o no. Las instrucciones que se pueden reusar son, por orden del fichero: Suma en doble precisión; Suma en simple precisión; Resta en doble precisión; Resta en simple precisión; Multiplicación en doble precisión; Multiplicación en simple precisión; División en doble precisión; División en simple precisión; Raiz cuadrada en doble precisión y Raiz cuadrada en simple precisión;

- `ISA_pos.conf`. Este fichero (también se puede modificar su nombre mediante `-f:pos`) contiene o bien una linea o bien tantas lineas como operaciones, con las reglas de acceso a la tabla de reuso para cada operación reusada. Si hay una sola linea, se supone que todas las operaciones tienen las mismas reglas de acceso. Cada linea debe contener 7 cifras. Las dos primeras contienen la mascara que permite decidir cuantos bits de cada operando hay que usar para acceder a la tabla. Por ej. si queremos usar 9 bits, la máscara será 1FF. La regla de acceso es, hacer la AND, desplazar a la derecha o a la izquierda los bits enmascarados (habitualmente no se desplazan) y a continuación hacer la XOR de ambos operandos junto con el último valor de la regla. (El acceso normal, para una tabla de 1K entradas, y asociatividad 2, es una regla "1FF 1FF 0 0 0 0 0").

- `ISA_acc.conf`. (Se puede modificar su nombre mediante `-f:acc`) Este fichero contiene una linea con 20 números enteros, pero es obsoleto y por tanto no tiene ningún efecto.

- `ISA_con.conf`. (Se puede modificar su nombre meidante `-f:con`) Fichero que indica si se debe aplicar la conmutatividad a las tablas de reuso. Contiene una secuencia de 10 valores booleanos (1 por operacion). Si dicho valor es verdadero, antes de acceder a la tabla de reuso el sistema ordenará los operandos de forma que valores iguales en distinto orden en una operación conmutativa no ocupen dos entradas en la tabla. Es una opción de configuración debido a que el hardware que realiza la operación no es trivial y en muchos casos es mejor no realizar esta comprobación ya que hay muy poca ganancia.

- `ISA_tol.conf`. (Se puede modificar su nombre mediante `-f:tol`) Fichero que contiene para cada operación (diez entradas) un entero que indica los bits tolerados al reusar dicha operación (OJO, el simulador no comprueba que este valor sea razonable o válido).

- `ISA_tot.conf`. (Se puede modificar su nombre mediante `-f:tot`) Fichero que contiene, para cada operación, la tolerancia en el caso de sumas o restas triviales (en los demás casos no se aplica su contenido). Si la configuración es de detectar de forma avanzada operaciones triviales (ver fichero `ISA_var` más adelante), este fichero determina que tolerancia hay

que aplicar a una suma antes de hacerla y dar por valido uno de los operandos. Esto tiene sentido desde el punto de vista de que una suma tolerante de dos operandos donde uno es mucho mayor que otro da como resultado el propio número mayor.

- `ISA_tab.conf`. (Se puede modificar su nombre mediante `-f:tab`) Este fichero contiene el número de tablas de reuso a utilizar y a que operaciones se asignan. Para ello contiene 10 enteros (uno por operación) que especifican a que tabla asignar cada operación. Si dos o más operaciones comparten una misma tabla el simulador realizará los cálculos consecuentemente.

- `ISA_cap.conf`. (Se puede modificar su nombre mediante `-f:cap`) Este fichero contiene para cada operación cuantos bits menos significativos de los operandos deben eliminarse antes de realizar la operación. Es importante notar que esto no es lo mismo que tolerar dichos bits, ya que si los bits se eliminan la operación será mal realizada aunque no se acierte en la tabla o no haya tabla. Esta opción permite simular unidades funcionales más cortas y ver los resultados.

- `ISA_var.conf`. (Se puede modificar su nombre mediante `-f:var`) Fichero de configuraciones varias. Contiene 8 enteros, cada uno con un significado específico:

  1. Asociatividad de la tabla. Este parámetro y la máscara del fichero ISA_pos, determinan el número de entradas de la tabla.
  2. Tipo de tabla usada. Admite tres valores: 0 para tablas finitas (reales), 1 para simular tablas infinitas y 2 para simular tablas finitas pero que solo memorizan las mantisas de las operaciones y calculan los exponentes.
  3. Detección de las operaciones triviales: 0 no activada, 1 activada y 2 deteccion avanzada, es decir, evita hacer sumas cuando los operandos son muy distantes.
  4. Tabla en acceso paralelo a la unidad aritmética (1) o secuencial con ella (0). En el primer caso siempre hay consumo de potencia en la unidad aritmetica, mientras que en el segundo el tiempo de cálculo en caso de fallo en la tabla aumenta.
  5. Activa el sistema de reuso tolerante consistente en guardar la media del resultado en lugar del resultado completo en la tabla. Como consecuencia la tabla de reuso es más estrecha. Con un valor falso no activa este sistema, mientras que un valor positivo indica cuantos bits adicionales (a los de los operandos) hay que guardar en el resultado.
  6. Activa o desactiva la tolerancia de las operaciones de comparación.
  7. Activa o desactiva el filtrado de operaciones. Con el filtrado activado hay que tener en cuenta otros ficheros.

8. Activa o desactiva el límite de reusos. Si el valor es un número positivo, además indica el número máximo de reusos de una entrada de la tabla antes de borrarla. Esta opción es útil para intentar limitar el error recurrente introducido por una misma operación.

3.2.1 Filtrado de operaciones.

Otra de las características del simulador es que permite filtrar operaciones. El filtrado de operaciones permite experimentar el reuso de instrucciones con profiling o dinámico. En el filtrado dinámico, la tabla de reuso trata de adaptarse al comportamiento del programa en tiempo de ejecución, mientras que con profiling, se realizan dos pasadas, en la primera se decide que instrucciones son útiles para reusar sobre el procesado de una imagen de muestra y en la segunda pasada, que ya simula una ejecución real, se utiliza el resultado de la primera pasada para intentar mejorar los resultados de reuso de otra imagen distinta.

Para poder realizar este filtrado, el uso es el siguiente: existen 4 posibles valores para la opción de filtrado (un 0, falso, significa que no se realiza filtrado de operaciones). Si la opción está activa, los posibles valores van del 1 al 4. Cada valor representa una fase del filtrado:

1. Con este tipo de filtro activado, solo ciertas instrucciones acceden a la tabla de reuso, realizandose un filtrado dinámico de instrucciones. Se define una tabla de filtro de tamaño la constante `TamFiltro` y la primera vez que una instrucción accede a la tabla se le asigna un valor de confianza (`MAXFIL`). Si una instrucción falla en la tabla de reuso `MAXFIL` veces seguidas no vuelve a acceder a la tabla, es decir, es un filtro que intenta eliminar las instrucciones que fallan siempre.

2. Este filtro, en cambio, establece un intervalo de confianza mediante un contador saturado de tamaño $\log_2$`MAXFIL`. El número de entradas del filtro también lo establece constante `TamFiltro`.

3. En esta opción, al igual que la siguiente, se utiliza para el filtrado con profiling. En este paso del proceso, concretamente, se realiza el filtrado usando el contenido del fichero `ISAFILTRO.conf`. Aquellas instrucciones cuyo PC no esté contenido en este fichero no accederán a la tabla de reuso.

4. Esta opción imprime una estadistica indicando que instrucción acaba de ejecutarse y si ha acertado o no. Un programa de ayuda, `filtrar.c` (ver anexo A) recoge esta salida y la transforma en un fichero que contiene para cada número de PC, la cantidad de hits y de accesos a las tablas de la instrucción. El fichero resultante puede ser procesado por cualquier método para conseguir un fichero con solo aquellos PC que contienen un porcentaje de aciertos aceptable. Dicho fichero deberá llamarse `ISAFILTRO.conf` y se usará en una posterior ejecución con la opción 3 de filtrado.

4 Conclusiones y extensiones futuras

El simulador SimpleReuse es un simulador basado en Simplescalar y Wattch que añade a estos simuladores 3 funcionalidades básicas:

1. Permite su configuración conjunta mediante un solo juego de ficheros de configuración y elimina la necesidad de recompilar los simuladores.
2. Permite la simulación de la capacidad de la CPU de reuso de instrucciones.
3. Permite la simulación de un sistema hardware de reuso de regiones.

El simulador puede ser todavía mejorado, incorporando principalmente dos líneas de análisis:

1. La capacidad de analizar, en tiempo real, regiones de instrucciones susceptibles de ser reutilizadas.
2. La incorporación de sistemas dinámicos de control del error.

A Código de filtrar.c

```
#include <stdio.h>

#define DESBALANCEO 5

typedef struct {
   long ins;
   long hits,accs;
   int amayor,amenor;
   void *mayor,*menor;
} elemento_arbol;

typedef elemento_arbol* ptr_elemento_arbol;

ptr_elemento_arbol inicio=NULL;
long ins=0;
int hit;

void ponerins(ptr_elemento_arbol punt, ptr_elemento_arbol ant)
{
   ptr_elemento_arbol temp;

   if (punt==NULL) {
      printf("Error, mal programado\n");
      exit();
   } else {
      if ((punt->ins)==ins) {
         punt->accs++;
         punt->hits+=hit;
      } else {
         if (ins>(punt->ins)) {
            if (punt->mayor==NULL) {
               punt->mayor=malloc(sizeof(elemento_arbol));
               if (punt->mayor==NULL) {
                  printf("Error, memoria agotada\n");
                  exit();
               }
               (punt->mayor)->ins=ins;
               (punt->mayor)->accs=1;
               (punt->mayor)->hits=hit;
               (punt->mayor)->amayor=0;
               (punt->mayor)->amenor=0;
               (punt->mayor)->mayor=NULL;
               (punt->mayor)->menor=NULL;
            } else {
```

```
      if (((punt->amayor)-(punt->amenor))>DESBALANCEO) {
         temp=punt->mayor;
         punt->mayor=temp->menor;
         temp->menor=punt;
         if (ant!=punt) {
            if (ant->mayor==punt)
               ant->mayor=temp;
            else
               ant->menor=temp;
         }
         punt->amayor=0;
         punt->amenor=0;
         if (inicio==punt) {
            inicio=temp;
            ant=temp;
         }
         ponerins(temp,ant);
      } else {
         punt->amayor++;
         ponerins(punt->mayor,punt);
      }
   }
} else {
   if (punt->menor==NULL) {
      punt->menor=malloc(sizeof(elemento_arbol));
      if (punt->menor==NULL) {
         printf("Error, memoria agotada\n");
         exit();
      }
      (punt->menor)->ins=ins;
      (punt->menor)->accs=1;
      (punt->menor)->hits=hit;
      (punt->menor)->amayor=0;
      (punt->menor)->amenor=0;
      (punt->menor)->mayor=NULL;
      (punt->menor)->mayor=NULL;
   } else {
      if (((punt->amenor)-(punt->amayor))>DESBALANCEO) {
         temp=punt->menor;
         punt->menor=temp->mayor;
         temp->mayor=punt;
         if (ant!=punt) {
            if (ant->mayor==punt)
               ant->mayor=temp;
            else
               ant->menor=temp;
```

```
                }
                punt->amayor=0;
                punt->amenor=0;
                if (inicio==punt) {
                   inicio=temp;
                   ant=temp;
                }
                ponerins(temp,ant);
             } else {
                punt->amenor++;
                ponerins(punt->menor,punt);
             }
          }
       }
    }
}

void Imprimir(ptr_elemento_arbol punt)
{
   if (punt->menor!=NULL) {
      Imprimir(punt->menor);
   }
   if (punt->mayor!=NULL) {
      Imprimir(punt->mayor);
   }
   printf("%ld %d %d\n",punt->ins,punt->hits,punt->accs);
}

void main()
{
   while (ins!=-1) {
      scanf("Instruccion: %ld %d\n",&ins,&hit);
      if (ins!=-1)
         if (inicio==NULL) {
            inicio=malloc(sizeof(elemento_arbol));
            inicio->ins=ins;
            inicio->hits=hit;
            inicio->accs=1;
            inicio->amayor=0;
            inicio->amenor=0;
            inicio->mayor=NULL;
            inicio->menor=NULL;
         } else {
            ponerins(inicio,inicio);
         }
```

```
   }

   if (inicio!=NULL)
      Imprimir(inicio);
   else
      printf("VACIO!!!!!!!!!!!!!\n");
}
```

B Ejemplos de ficheros de configuración.

B.1 Configuración para procesador 1 way en orden.

```
# random number generator seed (0 for timer seed)
-seed                            1

# instruction fetch queue size (in insts)
-fetch:ifqsize                   1

# extra branch mis-prediction latency
-fetch:mplat                     1

# speed of front-end of machine relative to execution core
-fetch:speed   1

# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred                       nottaken

# instruction decode B/W (insts/cycle)
-decode:width                    1

# instruction issue B/W (insts/cycle)
-issue:width                     1

# run pipeline with in-order issue
-issue:inorder                true

# issue instructions down wrong execution paths
-issue:wrongpath               true

# instruction commit B/W (insts/cycle)
-commit:width   1

# register update unit (RUU) size
-ruu:size                        4
```

```
# load/store queue (LSQ) size
-lsq:size                       2

# perfect memory disambiguation
#-lsq:perfect false

# l1 data cache config, i.e., {<config>|none}
-cache:dl1             dl1:128:32:4:f

# l1 data cache hit latency (in cycles)
-cache:dl1lat                   1

# l2 data cache config, i.e., {<config>|none}
-cache:dl2             none

# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1             il1:512:32:1:f

# l1 instruction cache hit latency (in cycles)
-cache:il1lat                   1

# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                     none

# flush caches on system calls
-cache:flush                false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress            false

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat               64 1

# memory access bus width (in bytes)
-mem:width                      4

# memory accesses are fully pipelined
#-mem:pipelined          false

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:16:4096:4:f

# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:32:4096:4:f

# inst/data TLB miss latency (in cycles)
```

```
-tlb:lat                        30

# total number of integer ALU's available
-res:ialu                       1

# total number of integer multiplier/dividers available
-res:imult                      1

# total number of memory system ports available (to CPU)
-res:memport                    1

# total number of floating point ALU's available
-res:fpalu                      1

# total number of floating point multiplier/dividers available
-res:fpmult                     1

# operate in backward-compatible bugs mode (for testing only)
-bugcompat                   false

# latencia de las operaciones:
-lat:IALU 1
-iss:IALU 1
-lat:IMUL 4
-iss:IMUL 1
-lat:IDIV 20
-iss:IDIV 19
```

B.2 Configuración para procesador 2 way en orden.

```
# random number generator seed (0 for timer seed)
-seed                           1

# instruction fetch queue size (in insts)
-fetch:ifqsize                  2

# extra branch mis-prediction latency
-fetch:mplat                    2

# speed of front-end of machine relative to execution core
-fetch:speed   1

# branch predictor type {nottaken|taken|perfect|bimod|2lev}
-bpred                       2lev

# instruction decode B/W (insts/cycle)
```

```
-decode:width                    2

# instruction issue B/W (insts/cycle)
-issue:width                     2

# run pipeline with in-order issue
-issue:inorder                true

# issue instructions down wrong execution paths
-issue:wrongpath               true

# instruction commit B/W (insts/cycle)
-commit:width   2

# register update unit (RUU) size
-ruu:size                        8

# load/store queue (LSQ) size
-lsq:size                        4

# l1 data cache config, i.e., {<config>|none}
-cache:dl1             dl1:128:32:4:f

# l1 data cache hit latency (in cycles)
-cache:dl1lat                    1

# l2 data cache config, i.e., {<config>|none}
-cache:dl2             ul2:256:64:4:l

# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1             il1:512:32:1:f

# l1 instruction cache hit latency (in cycles)
-cache:il1lat                    1

# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                     dl2

# flush caches on system calls
-cache:flush                 false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress             false

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat               64 1
```

```
# memory access bus width (in bytes)
-mem:width                        8

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:16:4096:4:f

# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:32:4096:4:f

# inst/data TLB miss latency (in cycles)
-tlb:lat                         30

# total number of integer ALU's available
-res:ialu                         2

# total number of integer multiplier/dividers available
-res:imult                        1

# total number of memory system ports available (to CPU)
-res:memport                      2

# total number of floating point ALU's available
-res:fpalu                        1

# total number of floating point multiplier/dividers available
-res:fpmult                       1

# operate in backward-compatible bugs mode (for testing only)
-bugcompat                    false

# latencia de las operaciones:
-lat:IALU 1
-iss:IALU 1
-lat:IMUL 3
-iss:IMUL 1
-lat:IDIV 20
-iss:IDIV 19
```

B.3 Configuración para procesador 4 way fuera de orden.

```
# random number generator seed (0 for timer seed)
-seed                             1

# instruction fetch queue size (in insts)
-fetch:ifqsize                    4
```

```
# extra branch mis-prediction latency
-fetch:mplat                     3

# branch predictor type {nottaken|taken|perfect|bimod|2lev}
-bpred                         comb

# bimodal predictor BTB size
-bpred:bimod                  2048

# 2-level predictor config (<l1size> <l2size> <hist_size>)
-bpred:2lev            1 1024 8 0

# instruction decode B/W (insts/cycle)
-decode:width                    4

# instruction issue B/W (insts/cycle)
-issue:width                     4

# run pipeline with in-order issue
-issue:inorder                 false

# issue instructions down wrong execution paths
-issue:wrongpath                true

# register update unit (RUU) size
-ruu:size                       16

# load/store queue (LSQ) size
-lsq:size                        8

# l1 data cache config, i.e., {<config>|none}
-cache:dl1             dl1:128:32:4:l

# l1 data cache hit latency (in cycles)
-cache:dl1lat                    1

# l2 data cache config, i.e., {<config>|none}
-cache:dl2             ul2:1024:64:4:l

# l2 data cache hit latency (in cycles)
-cache:dl2lat                    6

# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1             il1:512:32:1:l
```

```
# l1 instruction cache hit latency (in cycles)
-cache:il1lat                    1

# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                      dl2

# flush caches on system calls
-cache:flush                 false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress             false

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat               64 1

# memory access bus width (in bytes)
-mem:width                       8

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:16:4096:4:l

# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:32:4096:4:l

# inst/data TLB miss latency (in cycles)
-tlb:lat                        30

# total number of integer ALU's available
-res:ialu                        4

# total number of integer multiplier/dividers available
-res:imult                       1

# total number of memory system ports available (to CPU)
-res:memport                     2

# total number of floating point ALU's available
-res:fpalu                       4

# total number of floating point multiplier/dividers available
-res:fpmult                      1

# operate in backward-compatible bugs mode (for testing only)
-bugcompat                   false

# latencia de las operaciones:
```

```
-lat:IALU 1
-iss:IALU 1
-lat:IMUL 3
-iss:IMUL 1
-lat:IDIV 20
-iss:IDIV 19
```

References

[1] Doug Burger and Todd M. Austin. "The SimpleScalar Tool Set, version 2.0" in *Wisconsin-Madiso CS Dep. technical Report #1342*, 1997.

[2] David Brooks and Vivek Tiwari and Margaret Martonosi. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations" in *Proceedings of the 27th annual International Symposium on Computer Architecture*, (ISCA 00), 2000.